
# Unidad 7:Lenguaje procedimiental en SQL. Gestión de Cursores

## 7.1 Introducción

La mayoría de los SGBD comerciales, además del lenguaje que permite la ejecución de sentencias propias de SQL (DDL, DML y DCL), incluyen otras sentencias que permiten el uso de SQL como un lenguaje propio de programación, con elementos de programación estructurada tales como IF, WHILE, etc. Esto es, permiten la creación de scripts más avanzados.

En muchos SGBD, tales como TRANSACT-SQL (MS SQL Server) y Oracle, a este grupo de elementos se les llama PL/SQL.

Las sentencias de control de flujo son utilizadas por los precompiladores de los SGBD, lo que nos permitirá el uso de bloques SQL en un lenguaje anfitrión en lugar de sentencias simples.

Podremos, por tanto, desde el lenguaje anfitrión hacer llamadas a bloques más complejos de SQL, descargando así al primero de parte del procesamiento de datos. De esta forma el lenguaje anfitrión se puede encargar exclusivamente del interfaz mientras que la gestión de los datos será responsabilidad del SGBD.

Con esto último, estamos separando aún más los procesos y los datos, lo que optimiza el diseño de aplicaciones informáticas.

## 7.2 Tipos de datos en MySQL

Consulta los tipos de datos con los que puedes trabajar en la Unidad 4 o en el manual de referencia de MySQL (https://dev.mysql.com/doc/refman/8.0/en/data-types.html).

## 7.3 Trabajar con Variables.

Aunque ya hemos visto como trabajar con variables en unidades anteriores, vamos a repasarlo.

Una variable en MySQL Server es un objeto que contiene un valor individual de datos de un tipo específico. Normalmente, las variables se utilizan en lotes y secuencias de comandos:

- Como contadores, para contar el número de veces que se realiza un bucle o controlar cuántas veces debe ejecutarse.
- Para contener un valor de datos que se desea probar mediante una instrucción de control de flujo.
- Para guardar el valor de un dato que se va a devolver en un parámetro de retorno de un procedimiento almacenado o como el dato que se devuelve en una función.
- Etc.

Las variables en MySql pueden ser de usuario o de sistema.

### Variables de sistema

Las variables de sistema son todas aquellas que tiene definidas nuestro servidor de datos y que configuran la forma de trabajo de nuestro servidor.

El manejo de estas variables nos permitirán conocer el modo de trabajo de nuestro servidor y configurar y optimizar dicho servidor.

Podemos ver un listado de estas variables mediante la instrucción “SHOW”

→ show variables; /* mostrará un listado de las variables del sistema y su valor */

→ show variables like '%innodb%'; /* mostrará un listado de las variables del sistema relacionadas con el motor innodb */

## Variables de usuario

Las variables de usuario son aquellas que un usuario utiliza para almacenar valores que pueda utilizar posteriormente en sus bloques de programa. Estas variables pueden ser locales o de sesión.

Variables (de usuario) de sesión

En MySQL Server, podremos definir variables de ámbito global para la sesión en la que se definen.

Las variables de sesión **no se declaran.**

Se identifican porque comienzan por el símbolo "@".

Las utilizaremos para asignarles valores temporales que serán accesibles desde cualquier parte para la sesión (hilo) actual.

Cuando se cierre la sesión o para otra sesión establecida, aunque sea del mismo usuario, esta variable no tendrá el mismo valor.

Aunque no debemos abusar del uso de estas variables, su utilización a veces se hace imprescindible, por ejemplo en seguridad. En este caso se utilizan para garantizar desde cualquier parte de una aplicación que el usuario conectado realmente se ha autenticado.

Variables (de usuario) locales

Las variables locales en MySql deben declararse dentro de un bloque de código, es decir entre BEGIN y END, y justo después de BEGIN (antes de cualquier otro comando) para ello utilizamos la palabra reservada DECLARE:

Estas variables que declaramos, serán de ámbito local (como indica su nombre), es decir, no tendrán valor ni existirán fuera del bloque de código donde se definieron.

El tipo de datos de estas variables será cualquiera de los tipos de datos del sistema (todos los tipos de datos que ya hemos visto), también podrán declararse variables del tipo "CURSOR" (de este tipo de datos nos ocuparemos a lo largo de esta unidad).

**Sintaxis**

Declaración de variables locales

**DECLARE** nomvar1[, nomvar2, ...] tipodato1;

**DECLARE** nomvar3[, nomvar4, ...] tipodato2;

….

Nota.- Las variables locales siempre tendrán que ser declaradas al comienzo del cuerpo del procedimiento, después de BEGIN y antes de la primera sentencia del procedimiento o función. A esta zona del cuerpo de un procedimiento se le llama zona de declaraciones.

Nota.- Solo podremos utilizar DECLARE dentro del cuerpo de una rutina.

Cuando no estemos dentro de una rutina (procedimiento o función) solo podremos utilizar variables de sesión

Asignación de valores a variables (tanto locales como de sesión)

**SET** nombrevar = valor | expresión; - - Asignación de variable local

**SET** @nombrevar = valor | expresión; - - Asignación de variable de sesión

select columna1[, columna2,...] **INTO** nomvar1[, nomvar2, …]

from nomtabla

…

Mostrar resultado de variables

**SELECT** nomvar1[, nomvar2, …]

**SELECT** @nomvar

### 7.4 Bloques de programa.

Las instrucciones BEGIN y END se usan para agrupar varias instrucciones de MySQL en un bloque lógico.

El uso de las instrucciones BEGIN y END es obligatorio cuando:

En bucles WHILE/REPEAT que incluyan más de una instrucción.

En los elementos de una función CASE que incluya más de una instrucción.

En cláusulas IF o ELSE que incluyan más de una instrucción.

En el conjunto de instrucciones de las rutinas MySql (procedimientos o funciones).

Las instrucciones BEGIN y END deben usarse como un par: no se puede usar una sin la otra.

**Sintaxis**

BEGIN

Lista de {Sentencias sql|Sentencias de control de flujo}

*END*

Donde "{Sentencias sql|Sentencias de control de flujo }" Es cualquier instrucción o grupo de instrucciones MySQL válidos definidos como bloque de instrucciones.

### 7.5 Operadores en MySQL

**A.** Operador de asignación.- Asigna un valor escalar a una variable

Existen dos formas de asignar un valor a una variable:

a) Mediante el operador = prececido por SET:

**SET @mivar = 5**

**SET mivar = 5;** /* En una rutina, declarando la variable previamente */

b) Asignación del valor obtenido en una consulta:

```
SELECT numde into @mivar
FROM empleados
WHERE numem=100
```

**B.** Operadores aritméticos:

Suma ⇒ +

Resta ⇒ -

Multiplicación ⇒ *

División ⇒ /

Módulo ⇒ % (Resto de división entera)

División entera: DIV

Potencia ⇒ ^

**C.** Operadores de comparación:

Igual ⇒ =

Null – safe equal: <=>

Menor ⇒ <

Menor o igual ⇒ <=

Mayor ⇒ >

Mayor o igual ⇒ >=

Distinto ⇒ <>

**D.** Operadores lógicos:

ALL, ANY, SOME

AND, OR, NOT,

BETWEEN, EXISTS, IN, LIKE, RLIKE, REGEXP

**E.** Operador de concatenación de cadenas ==> No existe. Utilizamos la función CONCAT.

**F.** Comentarios en MySQL

a) En una sola línea: -- Texto de comentario
b) Varias líneas /* Texto en varias líneas */

### 7.6 Tablas temporales en MySQL

A veces, puede resultarnos útil almacenar de forma temporal ciertos datos en una tabla, pero no queremos que esta tabla forme parte de la estructura de nuestra base de datos, si no que después queremos que esta tabla desaparezca.

Esto lo podremos hacer mediante el uso de tablas temporales.

Para crear una tabla temporal utilizaremos la sintaxis de CREATE TABLE solo que indicaremos que es temporal:

```
CREATE TEMPORARY TABLE (…....);
```

El resto de la sentencia es idéntica a la de una tabla normal.

Una vez creada la estructura, podremos trabajar con ella con las sentencias de modificación de datos que vimos en la Unidad 5, como con cualquier otra tabla de la base de datos, la única diferencia es que el ámbito de nuestra tabla es la sesión donde ha sido creada, ni otros usuarios, ni el mismo usuario desde otra sesión tendrán acceso a ella.

Cuando cerremos la sesión esta tabla desaparecerá.

## 7.7 Constructores de control de flujo.

En MySQL podemos trabajar con las siguientes sentencias de control de flujo que nos permitirán que nuestras rutinas sean más complejas.

**1. CALL**

Lo utilizaremos para invocar y ejecutar un procedimiento almacenado que se habrá creado previamente mediante "CREATE PROCEDURE".

Sintaxis

```
CALL nombre_procedimiento ([lista_valores_param]);
```

**2. RETURN**

Termina la ejecución de una función devolviendo un valor.

Sintaxis

```
RETURN expresion;
```

"expresion" debe resolverse como un valor del tipo de datos que hayamos indicado que devuelve la función.

**3. Sentencia IF … ELSE**

Impone condiciones en la ejecución de una instrucción MySQL. La instrucción o bloque de instrucciones que siguen a la

palabra clave IF y a su condición se ejecuta si la condición se satisface, en caso contrario, se ejecuta la instrucción o bloque de instrucciones que siguen a la palabra ELSE (si existe, ya que esta es opcional).

Sintaxis

```
IF expresion_booleana THEN
[BEGIN]
  Sentencias SQL|Sentencias de control
[END]
[ELSE IF expresion_booleana THEN
[BEGIN]
  Sentencias SQL|Sentencias de control
[END]
]
[ELSE
[BEGIN]
  Sentencias SQL|Sentencias de control
[END]
]
        END IF [   ] /* habrá 1 línea 'end if' por cada aparición de  'IF' (también else if)*/
```

## 4. Sentencia CASE – Condiciones múltiples

Permite que la instrucción o conjunto de instrucciones que se ejecuten dependan de los distintos resultados de una expresión, o bien del resultado de distintas expresiones lógicas.

Tiene dos formatos:

**Función CASE sencilla**.- Compara una expresión con un conjunto de resultados para dicha expresión.

**Función CASE de búsqueda**.- Evalúa un conjunto de expresiones booleanas.

Ambos formatos aceptan el argumento ELSE opcional.

En ambos casos se ejecutará la primera instrucción o conjunto de instrucciones para las que se cumpla la expresión.

Sintaxis de sentencia CASE sencilla

```
CASE expresion
WHEN caso1 THEN rtdo1 - - cada rtdo. Puede ser un bloque de programa
WHEN caso2 THEN rtdo2
. . .
        [ELSE rtdoN]
END CASE;
```

Sintaxis de sentencia CASE de búsqueda

```
CASE
WHEN expres_bool1 THEN rtdo1
WHEN expres_bool2 THEN rtdo2
. . .
        [ELSE rtdoN]
END CASE;
```

La sentencia CASE **se puede utilizar en sentencias SELECT**.

Veamos un ejemplo con de cada formato:

**¡OJO! ==>** Dentro de SELECT en lugar de "END CASE", ponemos solo "END". Y no separamos cada opción con ";"

**FORMATO A (CASE sencilla).-** La expresión evaluada debe ser igual a alguno de los casos.

```
SELECT numem, nomem, ape1em, ape2em, - - leemos cuatro columnas de la tabla
CASE numhiem
        WHEN 0 THEN 'Cero hijos'
        WHEN 1 THEN 'Un hijo'
        WHEN 2 THEN 'Dos hijos'
        . . .
        WHEN 5 THEN 'Cinco hijos'
        ELSE 'Mas de cinco hijos'
        END as NUM_HIJOS,
    salarem, comisem - - leemos otras dos columnas de la tabla
FROM empleados;
```

En lugar de la columna numhiem, mostraremos un texto según el número de hijos

**FORMATO B** (case de búsqueda) ==> En cada caso se evalúa una expresión diferente.

```
SELECT numem, nomem, ape1em, ape2em,
        CASE
                WHEN numhiem= 0 THEN 'Cero hijos'
                WHEN  numhiem <= 3 THEN 'Entre 1 y 3 hijos'
                ELSE 'Mas de tres hijos'
END AS NUM_HIJOS,
 salarem, comisem
FROM empleados
```

### 5. Sentencia REPEAT – Bucles condicionales I

La sentencia o conjunto de sentencias que aparezcan dentro del cuerpo de esta instrucción se repetirá **hasta que** se cumpla una condición.

Sintaxis

```
REPEAT
[BEGIN]
   sentencias sql | sentencias de control
[END]
UNTIL condicion
END REPEAT;
```

### 6. Sentencia WHILE – Bucles condicionales II

La sentencia o conjunto de sentencias que aparezcan dentro del cuerpo de esta instrucción se repetirá **mientras** se cumpla una condición.

Sintaxis

```
WHILE condicion DO
[BEGIN]
   sentencias sql | sentencias de control
[END]
END WHILE
```

**Ejemplo//**

Duplicamos el precio de los productos hasta que la media del precio de dichos productos sea superior a 100.

Utilizando REPEAT

```
REPEAT
BEGIN
        UPDATE productos
        SET precio_unidad = precio_unidad*2
END
UNTIL (SELECT avg(precio_unidad) FROM productos) >=100
END REPEAT
```

Utilizando WHILE

```
WHILE (SELECT avg(precio_unidad) FROM productos) <100 DO
BEGIN
        UPDATE productos
        SET precio_unidad = precio_unidad*2
END
END WHILE
```

Las sentencias **Loop, Leave e Iterate** están en desuso, así que no se incluyen en esta unidad. Para más información consultar el **Manual de Referencia de MySQL**.

## 7.8 Control de excepciones. Manejadores de error en MySQL.

Las excepciones son situaciones que pueden producirse en tiempo de ejecución que no estaban previstas y que provocan errores inesperados.

Esto suele suceder cuando la consulta incluye una expresión que no es segura.

Debemos asegurarnos de que nuestro código provoque el menor número posible de estas excepciones, pero, a veces, es inevitable que se produzcan casos que no habíamos previsto.

Por esta razón los SGBD incluyen herramientas que nos permiten el control de este tipo de errores.

Es decir, cuando en una rutina se produce un error, de forma predeterminada, se detendrá la ejecución de la rutina y devuelve un error al bloque de código o a la aplicación que la llamó. Este resultado puede provocar situaciones inesperadas y descontroladas.

Sin embargo, si gestionamos bien los errores, las situaciones inesperadas pueden quedar controladas. Por ejemplo ejecutando un código que almacene el error, mostrando mensajes que clarifiquen cual ha podido ser el problema, continuando la ejecución si lo deseamos, etc.

En MySql Server, la forma de gestionar los errores es mediante la definición de MANEJADORES DE EXCEPCIONES en nuestras rutinas. Estos manejadores también se utilizan en los CURSORES en MySql Server.

Sintaxis de los Manejadores

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
        {SQLSTATE cod_estado_sql | MySQL codigo_error | nombre_error_usuario}
        acciones_error
```

CONTINUE | EXIT ==> Determina el tipo de manejador:

EXIT: Después de producirse el error saldremos del bloque de código donde se ha producido

CONTINUE: Después del error, continúa la ejecución por la siguiente instrucción a la que provocó el error.

SQLSTATE cod_estado_sql | MySQL codigo_error | nombre_error_usuario ==>

Determina el tipo de error que activará el manejador: Estado SQL, error MySQL o error definido por el usuario.

**Ejemplo//**

Procedimiento que inserta un empleado nuevo, en caso de que la clave primaria esté duplicada, avisará e insertará en la tabla control_errores una fila con el error producido y la fecha en la que se ha producido.

Supongamos que la estructura de control_errores es:

control_errores (cod_control(INT PK), fec_error (DATETIME), mensaje_error (VARCHAR(100))

```
DELIMITER $$
CREATE PROCEDURE inserta_empleado (cod_emp INT, nombre VARCHAR(100), ….)
MODIFIES SQL DATA
BEGIN
    DECLARE EXIT HANDLER FOR ‘1062’ | SQLSTATE ‘23000’
    BEGIN
        DECLARE control INT;
        SET control = (select max(cod_control)+1 FROM control_errores);
        INSERT INTO control_errores
        values (control,
                                    current_date, 'inserción clave duplicada en empleados');
    END;
    INSERT INTO empleados (cod_emp, nombre, ….);
    …
END;$$
```

**Ejercicio//** Crea la tabla de control de errores en la base de datos empresa. Averigua cual es el código de error para la restricción de integridad referencial (no existe el valor de clave foránea donde es clave primaria). Prepara un procedimiento almacenado que inserte un empleado nuevo, incluye el manejador de error para clave primaria repetida del ejercicio anterior y añade un manejador de error de restricción de integridad referencial con la tabla departamentos).

## 7.7. Concepto de CURSOR. Cursores en MySQL Server

Sabemos que, para obtener el conjunto de resultados que necesitamos de una o varias tablas de nuestra base de datos, utilizamos la sentencia SELECT.

Es decir, con SELECT obtenemos un conjunto de filas y columnas que contiene los datos que nos interesa en cada momento.

Pero a veces no basta con obtener ese conjunto de resultados, si no que lo que nos interesa es RECORRER FILA A FILA dicho conjunto de resultados y trabajar con los datos obtenidos en cada fila.

Para hacer este recorrido es para lo que vamos a utilizar LOS CURSORES.

Los cursores son estructuras pesadas, en cuanto a los recursos que consumen, por lo que los utilizaremos solo cuando sea necesario.

Todas las herramientas de gestión de bases de datos relacionales implementan esta estructura, y su manejo será similar. De una a otra cambiarán las características de dichos cursores.

Veámoslos en MySQL Server:

**Cursores en MySQL Server**

Los cursores en MySQL son muy simples:

Solo de lectura.

Solo podremos recorrerlos hacia delante y solo de una en una fila. Esto es:

Cuando lleguemos al final del conjunto de resultados no podremos volver a recorrerlo.

Si estamos en la fila 10 de mi cursor, solo podré ir a la fila 11.

**Sintaxis**

A) Declaración del cursor:

```
DECLARE nom_cursor CURSOR FOR sentencia_select;
```

B) Para poder trabajar con el cursor tendremos que abrirlo (llenar la estructura con las filas y columnas del mismo):

```
OPEN nom_cursor;
```

C) Para posicionarnos en la siguiente fila del cursor y utilizar los datos de cada celda (valor de columna):

```
FETCH nom_cursor INTO variable1, variable2, …;
```

Nota.- Tendremos que utilizar una variable para cada columna del cursor, del mismo tipo y en el mismo orden que en la declaración.

D) Al finalizar de trabajar con el cursor tendremos que cerrarlo:

```
CLOSE nom_cursor;
```

Los cursores se recorren dentro de bucles (REPEAT o WHILE).

Cuando llegamos al final de un cursor e intentamos posicionarnos en la siguiente fila, se produce un WARNING (SQLSTATE 02000), así que utilizaremos un manejador de error para este código y poder salir del bucle.

## 7.8. Bibliografía

MySQL 5.7 Reference Manual (http://dev.mysql.com/doc/refman/5.7/en/index.html)